Java For Beginners

Understanding Java Virtual Machine (JVM) Architecture

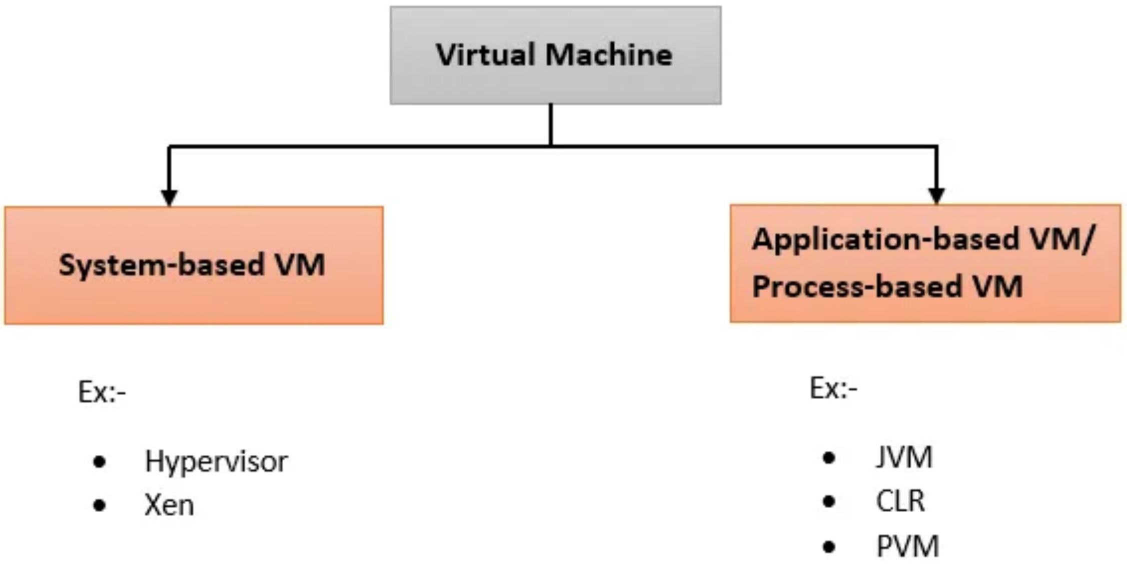
Jalitha Dewapura Follow 8 min read · May 10, 2021

163 3

Most Java developers and students who study software engineering don't understand what is the importance of learning basic concepts in Java. But understanding these basic concepts will make you an efficient programmer. This article will help you gain a solid understanding of how JVM works.

What is Virtual Machine?

Before jump into the JVM, let's discuss the concept of the virtual machine(VM). A virtual machine is some machine that physically does not exist. But VM will make the environment that you feel as it's real. VM can be an operating system or application environment that is installed on software. There are two categories of VM as shown below.

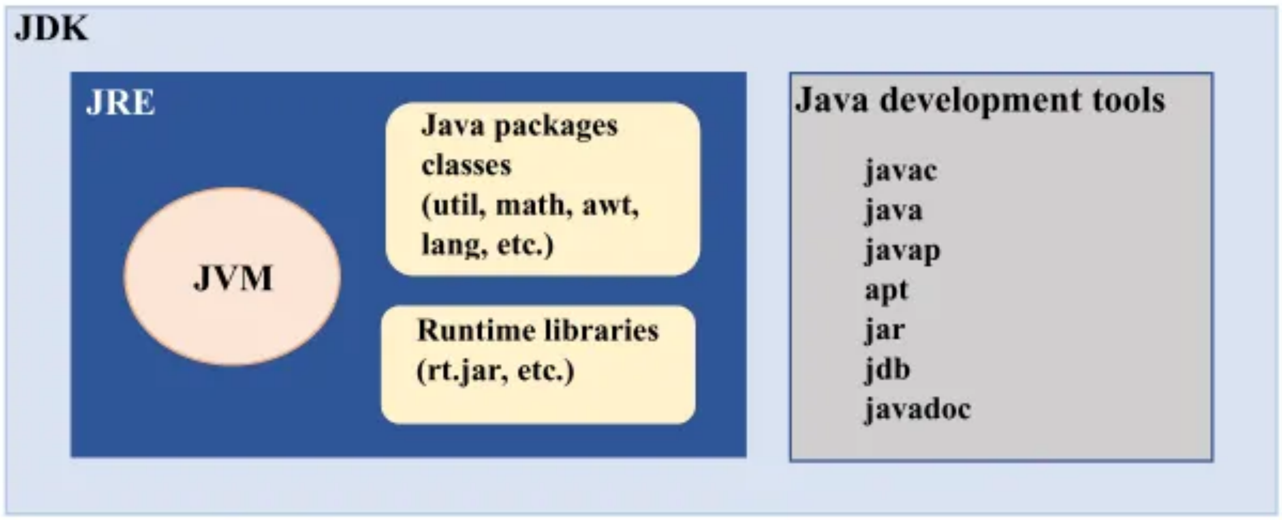


Types of virtual machine

System-based VM allows the sharing of the underlying physical machine resources (at least one hardware) between different virtual machines. But application-based VM does not have any hardware and it requires some application or software to create their virtual environment. That environment(platform) involves running some kind of a language and converting it into a different language. JVM is also under the application-based VM.

Java Architecture

Let's look at Java architecture to get a high-level picture of JVM.


JDK
JRE
JVM
Java packages classes (util, math, awt, lang, etc.)
Runtime libraries (rt.jar, etc.)
Java development tools
javac
java
javap
apt
jar
jdb
javadoc

Java Architecture

Let's discuss the difference between JDK, JRE, and JVM in brief.

Java Development Kit (JDK)

The **Java Development Kit (JDK)** is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development.

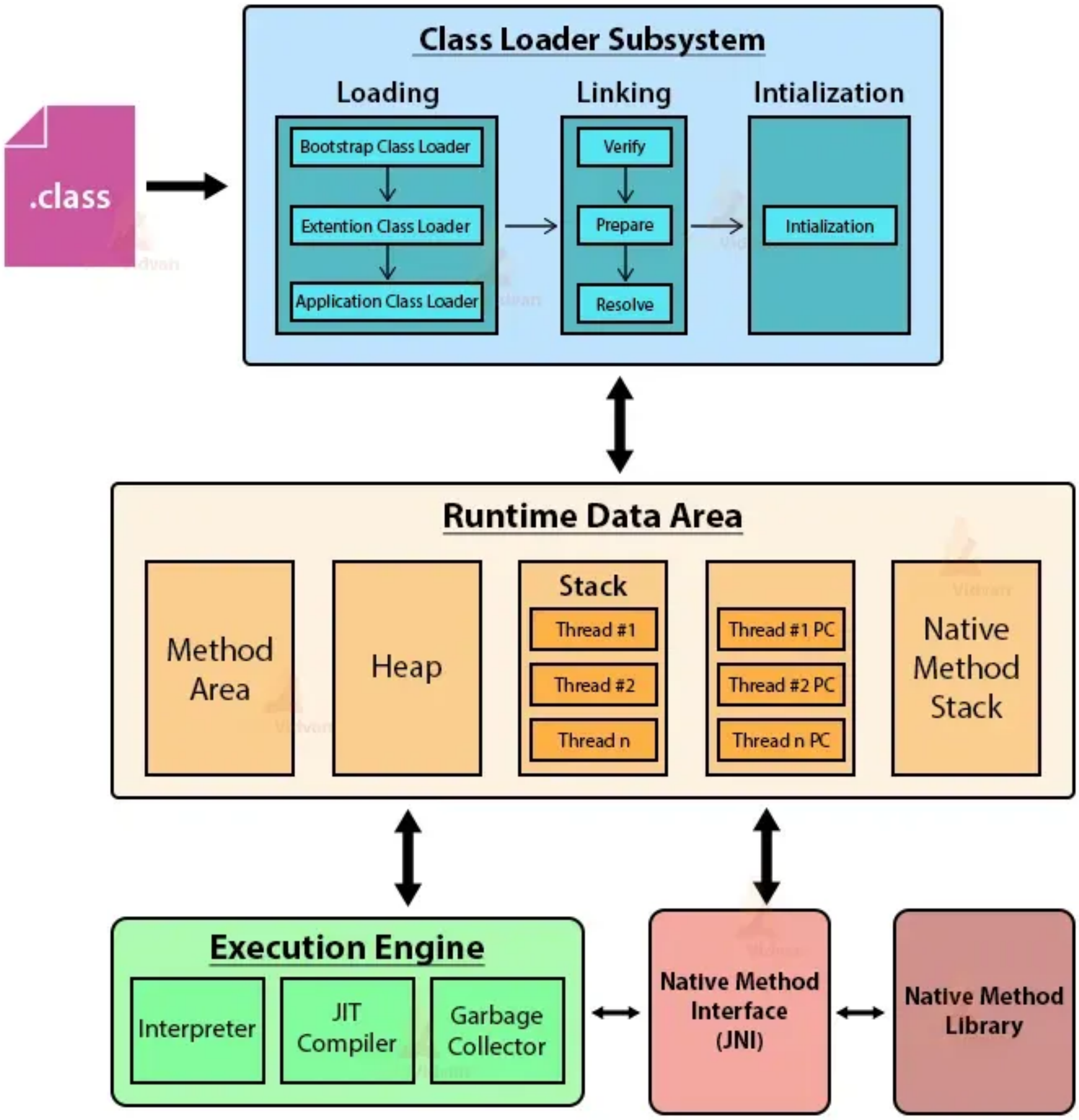
Java Runtime Environment (JRE)

The Java Runtime Environment provides the minimum requirements for executing a Java application. It consists of the Java Virtual Machine (JVM), java core packages, classes, and supporting files.

Java Virtual Machine (JVM)

The Java Virtual Machine is a specification that provides a runtime environment in which java bytecode can be executed. It means JVM creates a platform to run Java bytecode(.class file) and converting into different languages (native machine language) which the computer hardware can understand. Actually, there is nothing to install as JVM. When the JRE is installed, it will deploy the code to create a JVM for the particular platform. JVMs are available for many hardware and software platforms.

. . .



JVM Architecture

There are three main mechanisms inside the JVM as shown in the above diagram.

- ClassLoader

- Memory Area
- Execution Engine

ClassLoader

ClassLoader is responsible for loading class file into the memory area. The classloader is an abstract class, generates the data which constitutes a definition for the class using a class binary name which is constituted of the package name, and a class name. The classLoading mechanism consists of three main steps as follows.

- Loading
- Linking
- Initialization

Loading

Whenever JVM loads a file, it will load and read,

- Fully qualified class name
- Variable information (instance variables)
- Immediate parent information
- Whether class or interface or enum

When the class is loaded to the JVM, it creates an object of class type and is put into the heap area. This class type object will be created only the very first time the class is loaded to the JVM.

Linking

This is the process of linking the data in the class file into the memory area. It begins with **verification** to ensure this class file and the compiler.

1. Make sure this compiler is valid
2. The class file has the correct formating
3. The class file has the correct structure

When considering security threads, there is a possibility that some programs like malware can change or add some content to the class file. In this case, it will be identified by the byte code verifier and throw an exception called “verify exception”.

When the verification process is done, the next step is **preparation**. In this stage, all the variables are initialized with the default value. As an example, it will assign 0 for int variables, null for all objects, false for all boolean variables, etc.

Java allows you to use any domain-specific words in your java program. As an example, you can create a class by giving any name (except java keywords). But the JVM cannot understand these words. Therefore, JVM replaces these words from a memory address. That process is called **resolution**. At the end of these three steps, the java file is loaded to the memory area.

Initialization

This is the final stage of class loading. In this stage, the actual values are assigned to all static and instance variables. There is a rule that every class must be initialized before doing any active use. There are six active uses as follows.

1. Use `new` keyword.
2. Invoke static methods.
3. Assign values for static fields.
4. Initialize class.
5. Use `getInstance()` in Reflection API.
6. Instantiation of sub-class.

There are 4 ways to initialize a class in Java.

1. New Keyword: Goes through the normal initialization process.
2. Reflection API: `getInstance()` method and goes through the normal initialization process.
3. Clone Method: Gets the information from the source object.
4. IO.ObjectInputStream: Gets data from non-transient variables passed in the parameter.

Now you got a clear idea about the inside mechanism of the ClassLoader. There are three types of ClassLoaders as shown below.

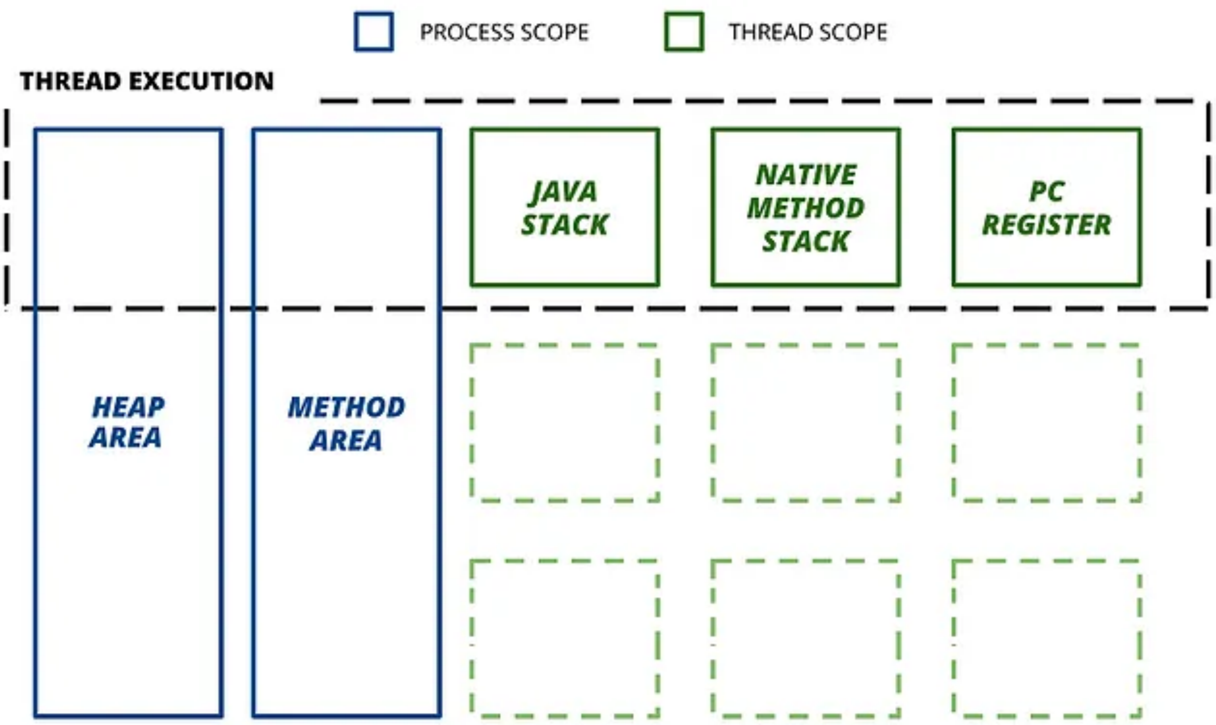
- Bootstrap ClassLoader — Load classes from JRE/lib/rt.jar

- Extension ClassLoader — Load classes from JRE/lib/ext
- Application/System ClassLoader — Load classes from classpath, -cp, Manifest

. . .

Memory Area

This is the place data will store until the program is executed. It consists of 5 major components as follows.



Memory Area

Method Area

This is a shared resource (only 1 method area per JVM). Method area stores class-level information such as

- ClassLoader reference
- Run time constant pool
- Constructor data — Per constructor: parameter types (in order)
- Method data — Per method: name, return type, parameter types (in order), modifiers, attributes
- Field data — Per field: name, type, modifiers, attributes

Heap Area

All the objects and their corresponding instance variables are stored in the heap area. As an example, if your program consists of a class called "Employee" and you are declaring an instance as follows,

```
Employee employee = new Employee();
```

When the class is loaded, there is an instance of `employee` is created and it will be loaded into the Heap Area.

Stack Area

There are separate stack areas for each thread. The stack is responsible for hold the methods (method local variables, etc) and whenever we invoke a method, a new frame creates in the stack. These frames use **Last-In-First-Out** structure. They will be destroyed when the method execution is completed.

Program Counter (PC) Register

PC Register keeps a record of the current instruction executing at any moment. That is like a pointer to the current instruction in a sequence of instructions in a program. Once the instruction is executed, the PC register is updated with the next instruction. If the currently executing method is 'native', then the value of the program counter register will be undefined.

Native Method Area

A native method is a Java method that is implemented in another programming language, such as C or C++. This memory area is responsible to hold the information about these native methods.

. . .

Execution Engine

At the end of the loading and storing mechanisms, the final stage of JVM is executing the class file. Execution Engine has three main components.

1. Interpreter
2. JIT Compiler
3. Garbage Collector

Interpreter

When the program is started to execute, the interpreter reads byte code line by line. This process will use some sort of dictionary that implies this kind of byte code should be converted into this kind of machine instructions. The main advantage of this process is the interpreter is very quick to load and fast execution.

But whenever it executes the same code blocks (methods, loops, etc) again and again, the interpreter executes repeatedly. It means the interpreter cannot be optimized the run time by reducing the same code repeated execution.

JIT Compiler

Just In Time Compiler (JIT Compiler) is introduced to overcome the main disadvantage of the interpreter. That means the JIT compiler can remember code blocks that execute again and again. As an example, there is a class called "Employee" and `getEmployeeID()` method is declared in this class. If a program uses `getEmployeeID()` method 1000 times, each time it is executed by the interpreter. But the JIT compiler can identify those repeated code segments and they will be stored as native codes in the cache. Since it is stored, next time JIT compiler uses the native code which stored in the cache.

Garbage Collector (GC)

As I already explained, all the objects are stored in the heap area before JVM starts the execution. . Since this area is limited, it is required to manage this area efficiently by removing the objects that are no longer in use. The process of removing unused objects from heap memory is known as **Garbage collection** and this is a part of memory management in Java. There are two ways the objects are considered as no longer in use.

1. When the reference is pointed to a null value.

```
Employee employee = new Employee();
employee = null;
```

Here the reference `employee` was pointing to the object of class 'Employee' and then a null is assigned to the reference. Since that moment, there is no reference pointing to the Employee object. Then this object is considered as a no longer reachable object.

2. When the reference is pointed to another object

```
Employee employee1 = new Employee(); //object 1
Employee employee2 = new Employee(); //object 2
employee1 = employee2;
```

Since the reference `employee1`is pointed to `employee2`, object 1 is considered as longer reachable.

Actually, the GC is a daemon thread that runs in the background. This process has two steps. First, the GC identifies the unused objects in memory (Mark), and then it removes the objects identified during the first step (Sweep).

Even though the garbage collection happens automatically, the programmer can trigger by calling `System.gc()` method.

. . .

Now I hope you have a clear understanding of JVM architecture. If I missed any points, let me know your suggestion in the comment. Specially thanks to Mr. Krishantha Dinesh who encouraged me to do this kind of article.

Reference

[1] Java Virtual Machine — CodeLabs

[2] Java Virtual Machine (JVM), Difference JDK, JRE & JVM - Core Java — Beginners Book

[3] JVM Tutorial — Java Virtual Machine Architecture Explained for Beginners

[4] Java Virtual Machine Working and Architecture — TechVidvan